The Report Committee for Jason Eric Trout
Certifies that this is the approved version of the following Report:

Testing Safety-Critical Systems using Model-Based Systems Engineering (MBSE)

APPROVED BY
SUPERVISING COMMITTEE:

Sarfraz Khurshid, Supervisor

Philip Terrall

Testing Safety-Critical Systems using Model-Based Systems Engineering (MBSE)

by

Jason Eric Trout

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2020

Dedication

To my wife, Katelyn, thank you for your love and support.

Acknowledgements

I would like to thank Dr. Sarfraz Khurshid for his guidance through many courses at The University of Texas at Austin as well as guidance throughout this report. I would also like to thank Philip Terrall for his mentorship and serving as a reader for this report.

Abstract

Testing Safety-Critical Systems using Model-Based Systems Engineering (MBSE)

Jason Eric Trout, M.S.E.

The University of Texas at Austin, 2020

Supervisor: Sarfraz Khurshid

Model-based Systems Engineering (MBSE) provides features for behavioral analysis, requirements traceability, system architecture, simulation, testing, and performance analysis that are imperative for the testing of safety-critical systems. In this report, we present a case study of a simple safety-critical system, and model the system using UML (Unified Modeling Language), SysML (Systems Modeling Language), and AADL (Architecture Analysis and Design Language). We then extend the AADL model with user-defined properties and annexes to augment additional analysis and reporting capabilities relevant to safety-critical systems. As safety and security expectations grow in concert with system complexity, MBSE will become increasingly ingrained in the workflow of the systems and software engineering communities.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

The typical system/software development life cycle (SDLC) consists of six stages: planning, analysis, design, implementation, integration/testing, and support/maintenance. In a waterfall-based SDLC, these stages are performed sequentially, with only one pass during the lifetime of a specific release of a product. In an iterative / agile-based SDLC, on the other hand, these stages are performed sequentially, many times during the lifetime of a specific release of a product, often in scheduled increments known as sprints.

A waterfall-based SDLC tends to be used more in the creation of safety-critical systems due to the rigor of analysis and testing requirements requisite for safety certification. Errors in analysis and design in the waterfall-based SDLC that are detected during the integration/testing phase tend to be more costly in monetary and scheduling terms than the more forgiving iterative / agile-based SDL, where errors can be detected and mitigated rapidly during subsequent product sprints.

Irrespective of the flavor of SDLC incorporated by an organization, after requirements are accepted and before an engineer begins implementing a system, modeling languages are often used during the design phase to document a system's architecture. The creation of modeling language-based artifacts is analogous to the creation of blueprints during the design phase used in manufacturing processes.

UML (Unified Modeling Language), created by Grady Booch, Ivar Jacobson, and James Rumbaugh at Rational Software in the early to mid-1990s, is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system (Rumbaugh et al., 2005).

UML provides structural diagrams that emphasize entities that must be present in a system, behavior diagrams that emphasize what must occur in a system, interaction diagrams that emphasize data and control flow amongst the things in the system, and implementation diagrams that show the structure of the run-time system.

Whereas the domain of the software engineer is in the implementation phase, the domain of the systems engineer is in the analysis and integration/testing phase. MBSE (Model-based systems engineering) extends modeling languages, such as UML, to make them useful within the domains of systems engineering. MBSE is expected to replace the document-centric approach that has been practiced by systems engineers in the past and to influence the future practice of systems engineering by being fully integrated into the definition of systems engineering processes (INCOSE, 2007). Due to integration/testing issues having higher cost in a waterfall-based SDLC, MBSE focuses on virtual integration. From the MBSE artifacts, systems engineers can integrate/test systems and perform analyses so that major architectural issues can be detected and mitigated before implementation begins. Two modeling languages that are enabling technologies for MBSE are SysML (Systems Modeling Language) and AADL (Architecture Analysis and Design Language).

SysML is a graphical modeling language that can be used to visualize and communicate the designs of sociotechnical systems on all scales (Delligatti, 2014). It is used as an architecture modeling language for systems engineering applications, and was created by the SysML Partners' SysML Open Source Specification Project in 2013. It was subsequently adapted and adopted by the Object Management Group (OMG) as OMG SysML in 2006. SysML is a superset of UML and supports the specification, analysis, design, and V&V of systems and systems of systems (SoS).

SysML introduces SysML extensions, namely: block definition diagrams (BDDs), which replaces the UML class diagrams, internal block diagrams (IBDs), that replaces the UML composite structure diagrams, requirements diagrams to document requirements, and parametric diagrams that permit the analysis of critical parameters. SysML-based tools such as Cameo Enterprise Architecture by Dassault Systèmes and Enterprise Architect by Sparx use SysML to allow simulation, testing, and requirements traceability of a system.

AADL is an architecture description language standardized by SAE (Society of Automotive Engineering). It focuses on system design specification using rich, formal semantics that can be used to analyze and generate systems (Delange, 2017). The architecture can be used either for documentation, for analysis, or for code generation. Its purpose is V&V and has an underlying specification language. It is more software-oriented and provides type primitives that can capture processes, threads, and data. AADL was designed for MBSE and has notation for specification of runtime architecture of safety-critical and secure software intensive systems.

Our goal in this report is to present a simple case study of a safety-critical system. We begin in the analysis phase, creating a Capability Requirements Specification (CRS) followed by a Software Requirements Specification (SRS). We start the design modeling process using UML and then step back to the analysis phase and show that the CRS and SRS could have been implemented in a modeling language as well—in this case SysML. We will integrate requirements and physical hardware specification to our system model.

SysML is a higher-level modeling language in contrast to AADL. AADL allows lower-level modeling that is typically required of real-time systems. The primary work in this report is creating AADL user-defined properties and AADL annexes to augment the

capabilities of AADL for the design of real-time, safety-critical systems. We will perform a SWaP (Size, Weight, and Power) analysis on the internal components of a signal processing unit and perform security analysis by creating an attack tree to show possible attack paths for our system.

Chapter 2: Case Study: Specifications and Initial UML Model

Our case study will involve a safety-critical ADS-B (Automatic Dependent Surveillance-Broadcast) signal processing unit that will serve as our SUT (System Under Test). The signal processing unit is part of a greater SoS that serves a hypothetical customer's analysis needs.

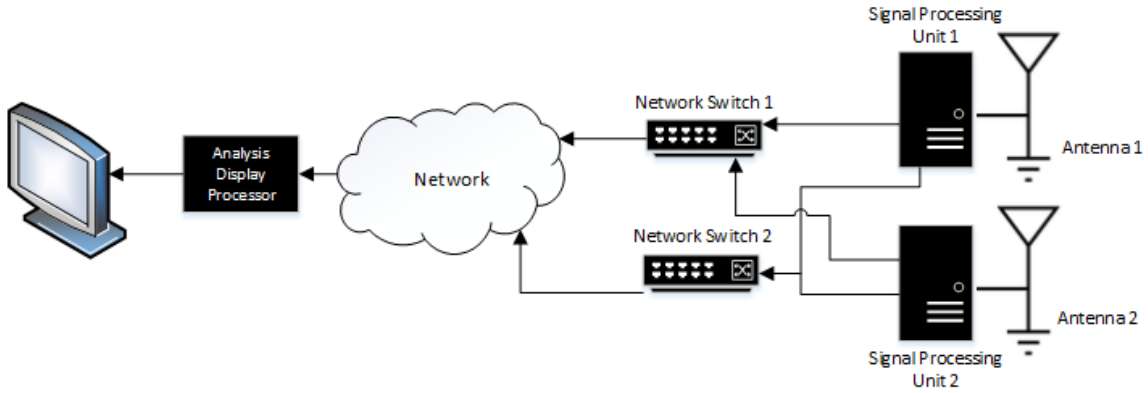

Figure 1: Case Study: System of Systems (SoS) / Data Flow

We will be using the traditional waterfall-based SDLC. Our focus within this case study is on the analysis stage. During the analysis phase, the customer-given capability requirements specification and engineering-derived software requirements specification are as follows:

Requirement ID	Requirement
CRS-1	The signal processing unit shall operate within an environment where temperatures are within the range of 26°C to 50°C inclusive.
CRS-2	The signal processing unit shall not exceed a width of 10cm.
CRS-3	The signal processing unit shall not exceed a height of 25cm.
CRS-4	The signal processing unit shall not exceed a depth of 20cm.

CRS-5	The signal processing unit's internal components shall not exceed a weight of 2kg.
CRS-6	The signal processing unit's internal components shall not exceed a power of 300W.
CRS-7	The signal processing unit shall transmit extracted ADS-B information over an existing network to an Analysis Display Processor.

Table 1: Case Study: Capability Requirements Specification

Requirement ID	Requirement
SRS-1	The signal processing unit shall not exceed a temperature of 27°C. (Links to CRS-1).
SRS-2	The signal processing unit shall control redundant fans. (Links to CRS-1).
SRS-3	The signal processing unit shall receive ADS-B at 1090MHz using mode-S extended squitter of the SSR transponder, with 50KHz of bandwidth. (Links to CRS-7).
SRS-4	The signal processing unit shall receive ADS-B at 978MHz (UAT), with 1.3MHz of bandwidth. (Links to CRS-7).
SRS-5	The signal processing unit shall process ADS-B signals within an 8ms time window. (Links to CRS-7).
SRS-6	The signal processing unit shall transmit extracted ADS-B information using the User Datagram Protocol (UDP) transport. (Links to CRS-7).

Table 2: Case Study: Software Requirements Specification

From the above SRS, the traditional next step in our analytical progression is to model the software using a modeling language. We will begin using UML, noting some pitfalls with respect to serving the needs of the systems engineering organization with discussion of how additional modeling languages, such as SysML and AADL can be used to mitigate these pitfalls.

We will create the following diagrams: (1) Signal Processing Unit Use Case Diagram, (2) Signal Processing Unit Class Diagram, (3) Maintain Temperature Sequence Diagram, (4) Maintain Temperature Activity Diagram, (5) Process Signal Sequence Diagram, and (6) Process Signal Activity Diagram.

A use case diagram represents a user's or system's interaction with the system at a high-level. For our case study, we have an actor named System Manager that interacts with the signal processing unit to maintain temperature and process signals.

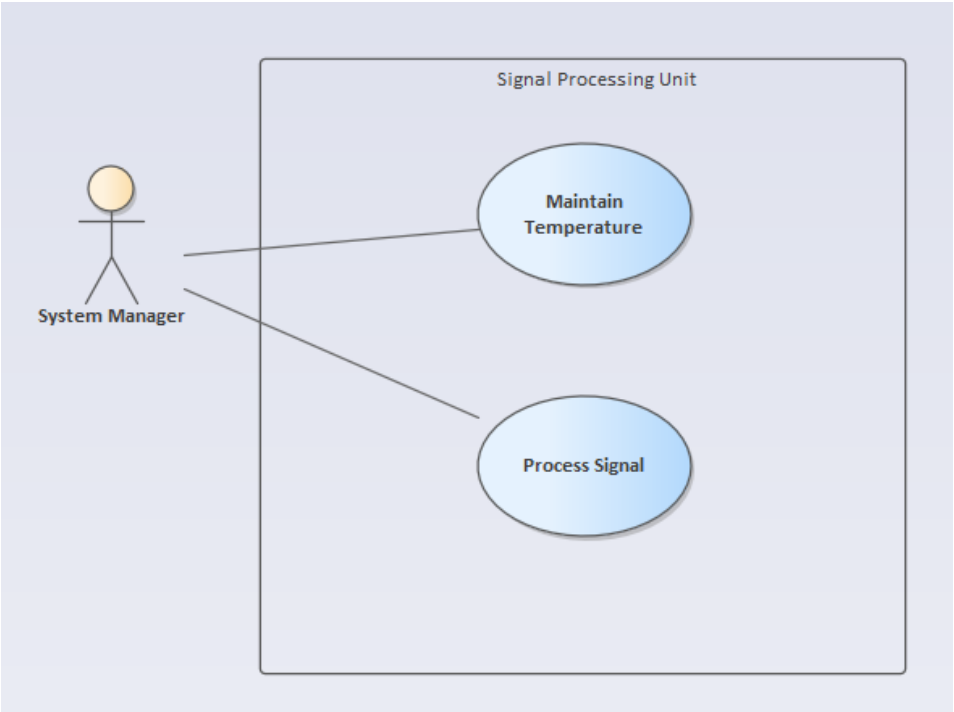

Figure 2: Case Study: Signal Processing Unit UML Use Case Diagram

We also make a signal processing unit class diagram. We have three classes: SystemManager, TemperatureMaintainer, and SignalProcessor. The SystemManager delegates method invocations to the TemperatureMaintainer and SignalProcessor classes.

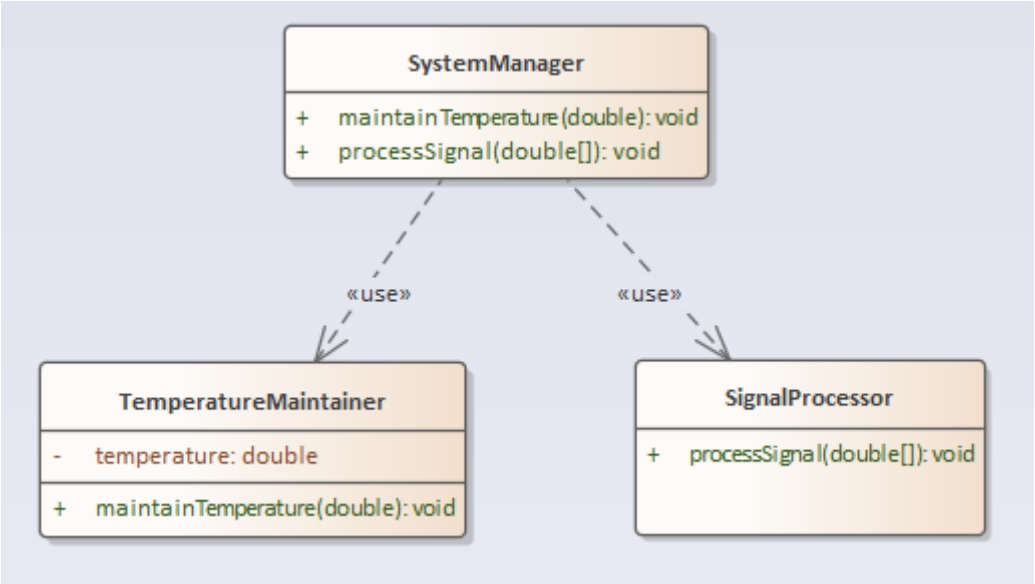

Figure 3: Case Study: Signal Processing Unit UML Class Diagram

The following sequence diagram shows the call flow from the System Manager actor for maintaining temperature.

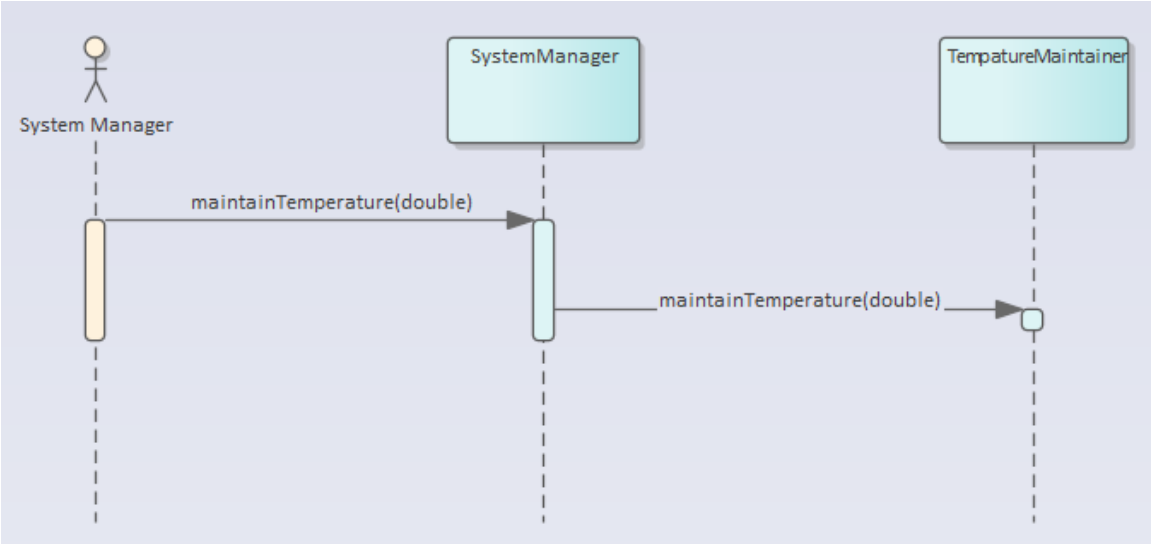

Figure 4: Case Study: Maintain Temperature UML Sequence Diagram

The following activity diagram shows the actions of the TemperatureMaintainer connected by control flows that indicate the sequence in which actions are fired. Our specification states that the temperature must be maintained to be less than or equal to 27°C. Based on our analysis, as long as a single fan is at the following capacities for the respective temperature ranges, we will be able to meet the set point per the specification. Two fans are controlled and utilized in case one fan fails.

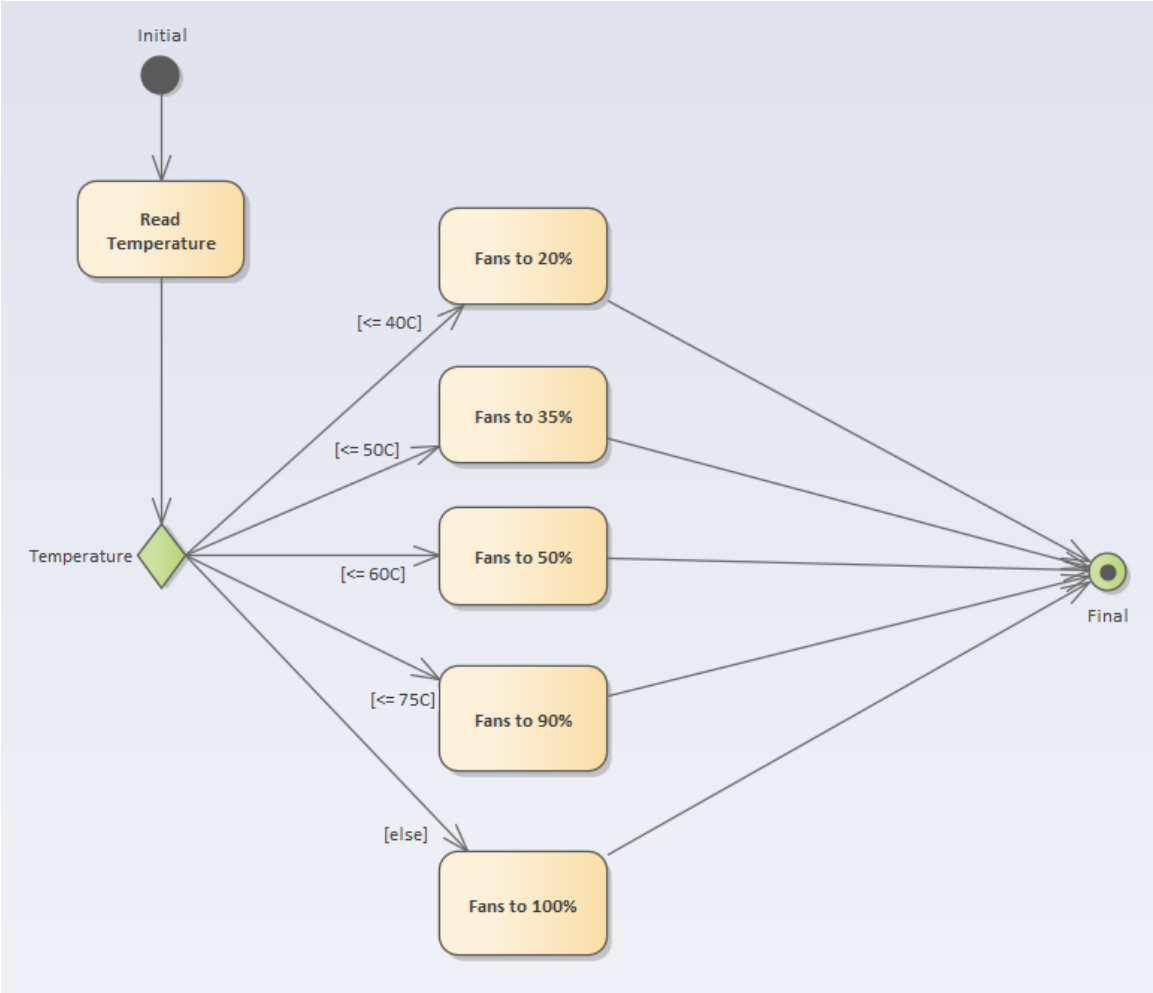

Figure 5: Case Study: Maintain Temperature UML Activity Diagram

The following sequence diagram shows the call flow from the System Manager actor for processing ADS-B signals.

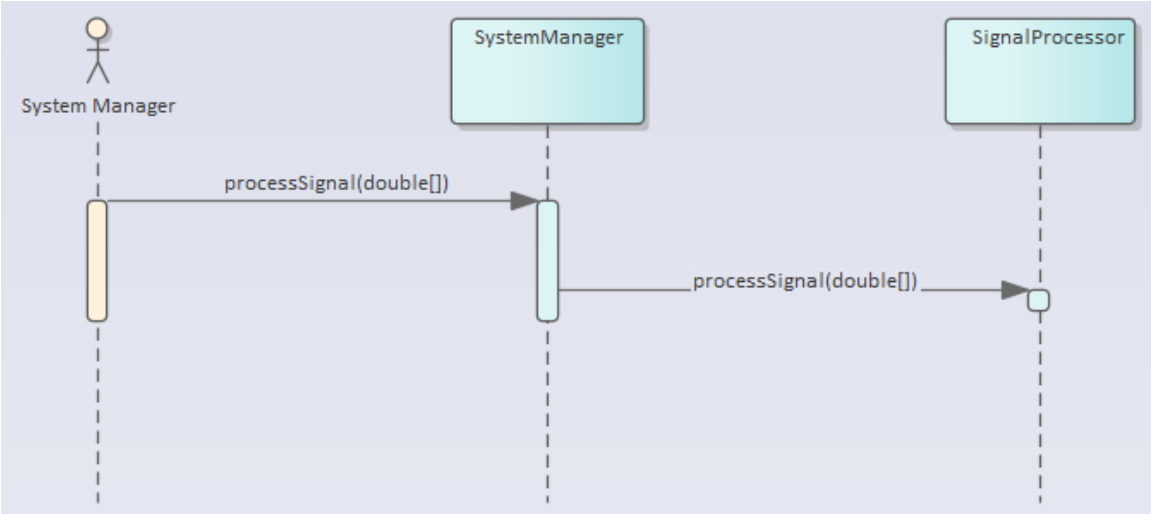

Figure 6: Case Study: Process Signal UML Sequence Diagram

The following activity diagram shows the actions of the SignalProcessor connected by control flows that indicate the sequence in which actions are fired. Our specification states we shall process ADS-B signals at both 978MHz and 1090MHz.

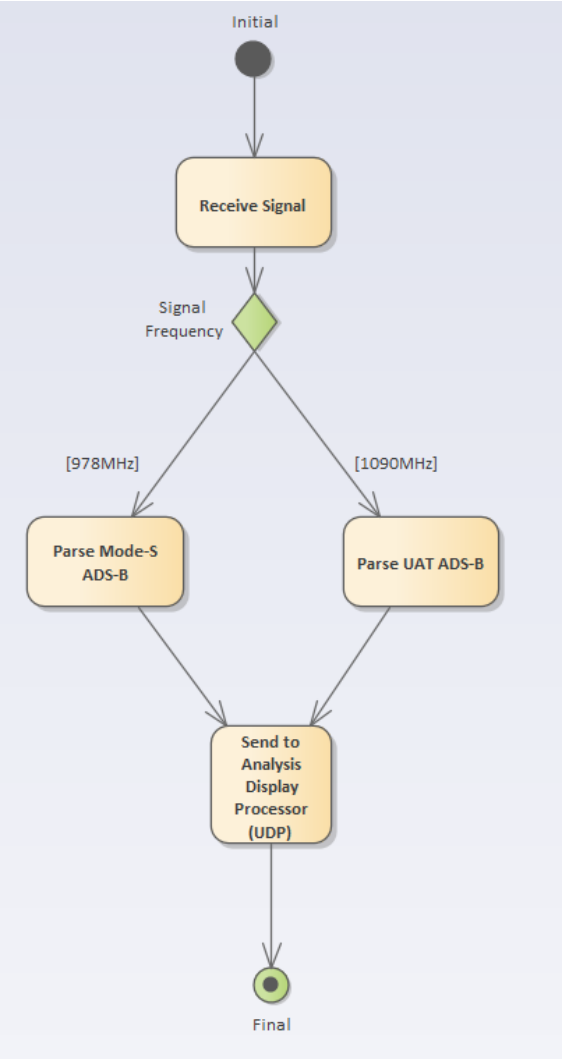

Figure 7: Case Study: Process Signal UML Activity Diagram

We have produced UML models that will be helpful for software engineers during the implementation phase; however, there are relevant systems engineering details not representable with UML. What are the constituent hardware components? What are the requirements? In the next chapter, we will use SysML to address these issues.

Chapter 3: Case Study: SysML Model

In the previous chapter, we created a simple UML model to model the software for the signal processing unit. We will now look how we could have augmented these software models with SysML models to address systems engineering activities during the analysis phase. We will create the following SysML diagrams: (1) Signal Processing Unit SysML Requirements Diagram, and (2) Signal Processing Unit SysML Block Definition Diagram. All the diagrams created in the previous chapter are valid SysML as well. We will not repeat them again in this chapter.

SysML provides a requirements diagram that can be used for capturing requirements within a system model. Capturing requirements in a system model in contrast to keeping requirements in a separate system has several benefits. The three most prominent benefits are that (1) requirements can easily be traced to design-level artifacts, (2) requirements do not get out of synchronization with design, and (3) requirements within the system model can still be exported to traditional document artifacts as necessary. We will take the original document-based CRS and SRS requirements from the previous chapter and place them in our system model. SRS requirements will be requirement entities that are associated to CRS requirements through a derive relationship. This allows the systems engineer to observe CRS and SRS requirements within the same model without having to trace through cumbersome links in traditional systems engineering tools.

SysML environments provide a way to trace requirements down to design-level artifacts and provide checklist functionality and completion analysis that are useful for providing requirements coverage information to stakeholders.

The requirements diagram is as follows:

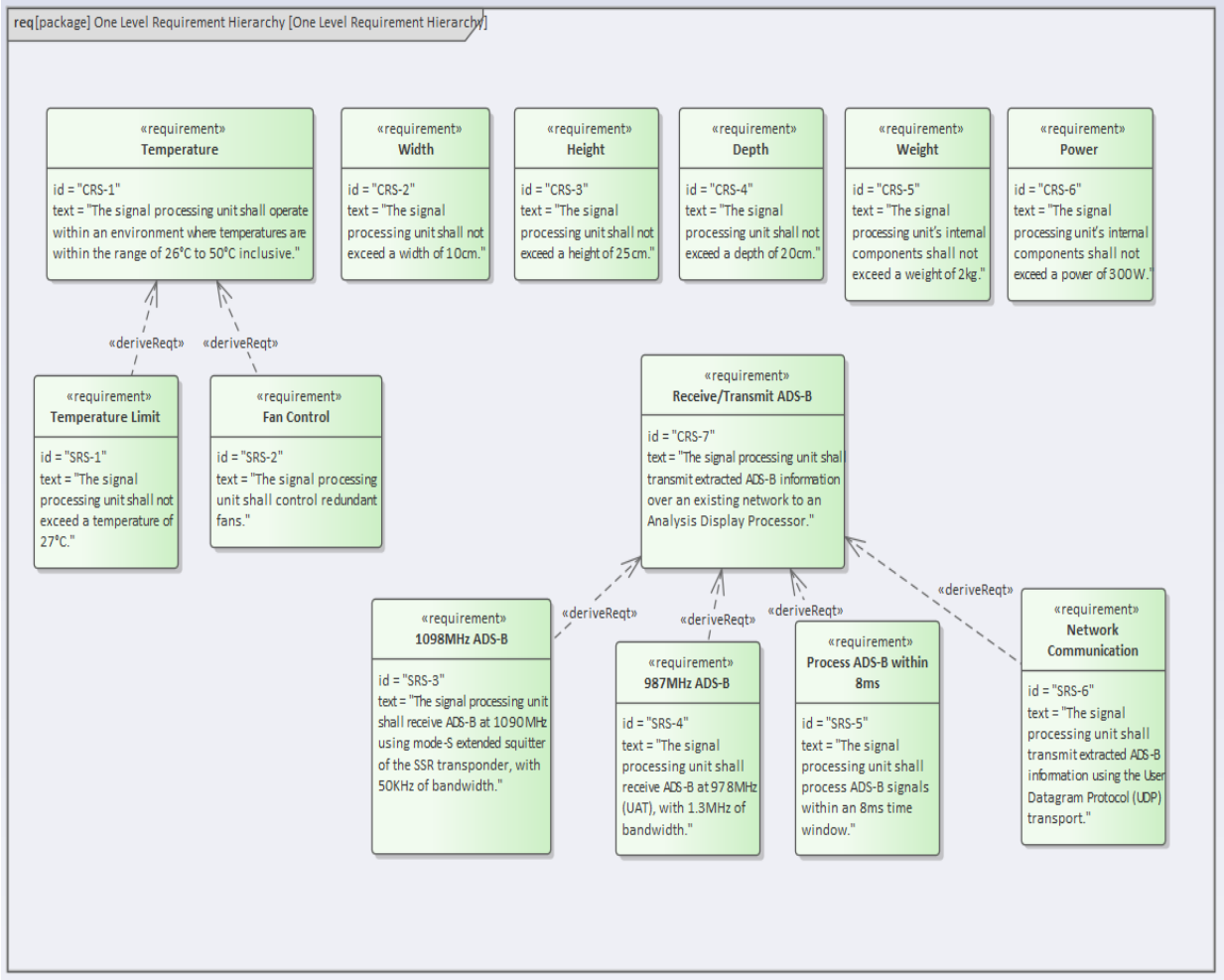

Figure 8: Case Study: Signal Processing Unit SysML Requirements Diagram

BDDs (Block Definition Diagrams) replace UML class diagrams in SysML. BDDs represent a system component such as software or hardware. In our case study, we will use a BDD to represent the physical hardware of the signal processing unit. We represent our system as well as its constituent parts as blocks. The parts that make up the system are represented with composition notations. We also utilize proxy ports to model

the interfaces with the external boundary of our system. Proxy ports differ from full ports in that they are not fully realized—this permits binding of realized physical ports later on in the analysis process (i.e. if use of a copper-based vs. fiber-based network medium has not yet been established).

The BDD for the signal processing unit is as follows:

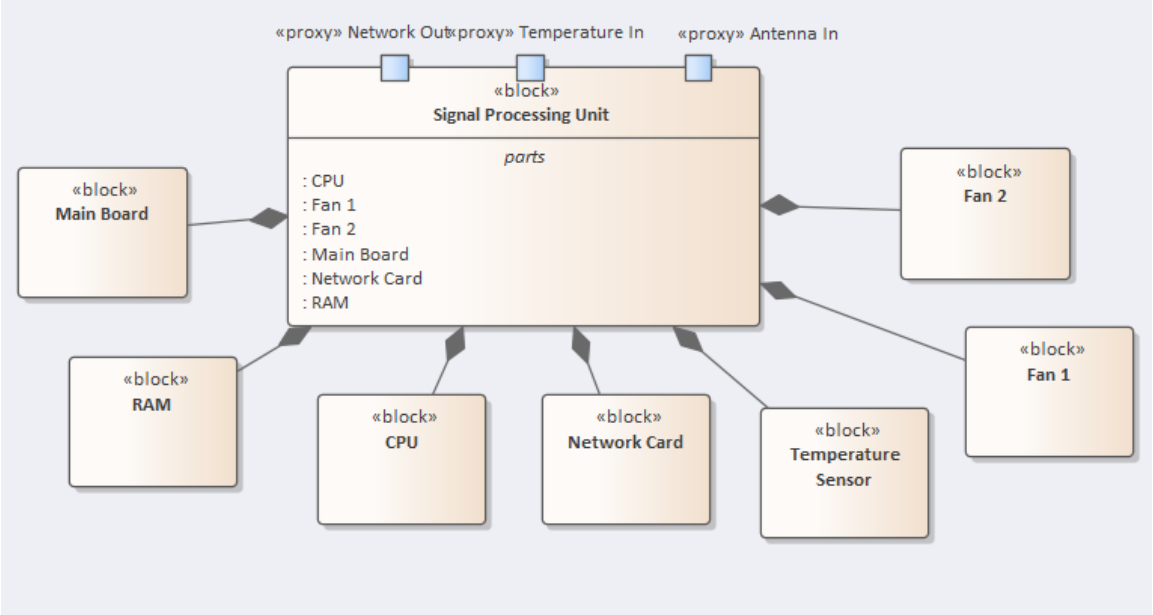

Figure 9: Case Study: Signal Processing Unit SysML BDD

We have seen that SysML, a superset of UML, allows modeling of systems engineering activities in a common way to the modeling of software by software engineers. Systems and software engineers can use the models jointly during the various phases of the SDLC. What can we do about lower-level requirements, such as timing requirements? In the next chapter, we will look at using AADL for this type of analysis.

Chapter 4: Case Study: AADL Model

We have a requirement specified in our SysML requirements diagram, identified as SRS-5, that states the signal processing unit shall process ADS-B signals within an 8ms time window. AADL is an architecture and analysis design language created for modeling real-time systems. It is best suited for modeling processors, memory, processes, and threads; consequently, it is the ideal modeling language to use for this requirement. We will use an open source IDE called OSATE that is based on Eclipse for our development of our AADL model.

In a similar way to modeling our signal processing unit as a SysML BDD in the previous chapter, AADL allows us to define devices for our system.

The devices.aadl file containing the devices package is as follows:

```
package devices
public

    device chassis
    end chassis;

    device main_board
    end main_board;

    processor cpu
    end cpu;

    memory ram
    end ram;

    device network_card

    end network_card;

    device temperature_sensor
    end temperature_sensor;

    device fan
    end fan;

end devices;
```

Figure 10: Case Study: AADL devices.aadl

We also create a system implementation that associates instances to these devices. This system implementation represents an instantiable model that can be analyzed. For

SRS-5, we are specifically interested in performing a latency analysis of the SignalProcessor process data flows.

The system implementation is as follows:

```
package integration
public
    with devices;
    with processes;
    system integration
    end integration;
    system implementation integration.functional
        subcomponents
            chassis: device devices::chassis;
            mb: device devices::main_board;
            ram1: memory devices::ram;
            nc: device devices::network_card;
            temp_sensor: device devices::temperature_sensor;
            fan1: device devices::fan;
            fan2: device devices::fan;
            signalProcessor: process processes::SignalProcessor.i;
    end integration.functional;
end integration;
```

Figure 11: Case Study: AADL integration.aadl

The subcomponents of the integration package contain the devices that were represented in our signal processing unit SysML BDD. We also added a SignalProcessor process implementation. This process has three threads: ReceiveSignalThread, ParseADSBThread, and SendADSBInfoToAnalysisDataProcessorThread. The SignalProcessor receives signal information as 112 bits of raw data and outputs a formatted ADS-B message that is 1KiB in size. We specify the individual thread latency ranges based on empirical analysis.

The AADL definitions of the SignalProcessor process is as follows:

```
with Data_Model;

data bits112
    properties
        Data_Size => 1 Bits;
        Data_Model::Dimension => (112);
end bits112;

data adsbmsg
    properties
        Data_Size => 1 Bytes;
        Data_Model::Dimension => (1024);
end adsbmsg;
```

Figure 12: Case Study: AADL data definitions (processes.aadl)

```
thread ReceiveSignalThread
    features
        dataIn : in event data port bits112;
        dataOut : out event data port bits112;
    flows
        flowThruReceiveSignalThread : flow path dataIn -> dataOut;
    properties
        latency => 1ms .. 2 ms applies to flowThruReceiveSignalThread;
end ReceiveSignalThread;

thread ParseADSBThread
    features
        dataIn : in event data port bits112;
        dataOut : out event data port adsbmsg;
    flows
        flowThruParseADSBThread : flow path dataIn -> dataOut;
    properties
        latency => 1ms .. 2ms applies to flowThruParseADSBThread;
end ParseADSBThread;

thread SendADSBInfoToAnalysisDataProcessorThread
    features
        dataIn : in event data port adsbmsg;
        dataOut : out event data port adsbmsg;
    flows
        flowThruSendADSBInfoToAnalysisDataProcessorThread : flow path dataIn -> dataOut;
    properties
        latency => 1ms .. 4ms applies to flowThruSendADSBinfoToAnalysisDataProcessorThread;
end SendADSBInfoToAnalysisDataProcessorThread;
```

Figure 13: Case Study: AADL thread definitions (processes.aadl)

```
process SignalProcessor
end SignalProcessor;

process implementation SignalProcessor.i
    subcomponents
        rst: thread ReceiveSignalThread;
        pat: thread ParseADSBThread;
        saitadpt: thread SendADSBInfoToAnalysisDataProcessorThread;
    connections
        incoming: port rst.dataOut -> pat.dataIn;
        outgoing: port pat.dataOut -> saitadpt.dataIn;
    flows
        pf: end to end flow rst.flowThruReceiveSignalThread ->
         incoming -> pat.flowThruParseADSBThread -> outgoing ->
          saitadpt.flowThruSendADSBInfoToAnalysisDataProcessorThread;
    properties
        latency => 3ms .. 8ms applies to pf;
end SignalProcessor.i;
```

Figure 14: Case Study: AADL SignalProcessor definitions (with latencies)

We use the OSATE analysis tools to perform a latency analysis of the signal processing unit. The generated latency analysis report is as follows:

1	Latency analysis with preference settings: AS-MF-DL-EQ-EQL							
2								
3	Latency results for end-to-end flow 'signalProcessor.pf' of system 'integration.functional'							
4								
5	Result	Min Specified	Min Actual	Min Method	Max Specifie	Max Actual	Max Method	Comments
6	thread signalProcessor.rst	1.0ms	1.0ms	specified	2.0ms	2.0ms	specified	
7	connection rst.dataOut -> pat.dataIn	0.0ms	0.0ms	no sampling/	0.0ms	0.0ms	no sampling/queuing latency	
8	thread signalProcessor.pat	1.0ms	1.0ms	specified	2.0ms	2.0ms	specified	
9	connection pat.dataOut -> saitadpt.dataIn	0.0ms	0.0ms	no sampling/	0.0ms	0.0ms	no sampling/queuing latency	
10	thread signalProcessor.saitadpt	1.0ms	1.0ms	specified	4.0ms	4.0ms	specified	
11	Latency Total	3.0ms	3.0ms		8.0ms	8.0ms		
12	Specified End To End Latency		3.0ms			8.0ms		
13	End to end Latency Summary							
14	INFO	Minimum actual latency total 3.00ms is greater or equal to expected minimum end to end latency 3.00ms						
15	INFO	Maximum actual latency total 8.00ms is less or equal to expected maximum end to end latency 8.00ms						

Figure 15: Case Study: AADL Latency Analysis Report

We can see from the latency report that based on our empirical timings for each thread in the SignalProcessor process, the minimum latency is 3ms; whereas, the maximum latency is 8ms; hence, the SignalProcessor process is in compliance with the SRS-5 requirement.

OSATE provides many analysis tools, including fault tree, bus load, and scheduling analysis. In the next chapter, we will look at implementing additional tools for OSATE, namely a SWaP Analysis Report tool and a security tool for generating attack trees that represent potential attack paths within our system.

Chapter 5: Extending AADL for Safety Analysis

AADL models can be extended beyond the core AADL language by use of user-defined properties and annexes. User-defined properties can be specified directly within the core AADL language, while annexes are more complex and require custom language parser implementation. We will use both types of AADL extensions with respect to our case study. All source code for these extensions can be found at https://github.com/jasontrout/AADL_extensions.

AADL User-Defined Properties for Generating A SWaP Analysis Report

Embedded computing manufacturers often strive to reduce SWaP with the goal of creating performant systems with minimal resource footprint. For safety-critical systems, violating SWaP requirements can result in system instability and general vulnerability. When performing an architectural analysis, it is imperative that we can perform trades amongst subcomponents with respect to SWaP easily and accurately. The user-defined properties functionality of AADL provides a means to achieve this goal.

In our case study, we have the following requirements: (1) The signal processing unit shall not exceed a width of 10cm, (2) The signal processing unit shall not exceed a height of 25cm, (3) The signal processing unit shall not exceed a depth of 20cm, (4) The weight of the internal components of the signal processing unit shall not exceed 2kg, and (5) The power consumed by the internal components of the signal processing unit shall not exceed 300W. We will create a property set called swap_properties used exclusively for SWaP analysis.

The property set swap_properties is defined in a file named swap_properties.aadl.

```
property set swap_properties is
  size_units: type units (cm, m => cm * 100, mm => m * 0.001, inches => cm * 16.387064);
  width: aadlreal units swap_properties::size_units applies to (system, device, processor, memory);
  height: aadlreal units swap_properties::size_units applies to (system, device, processor, memory);
  depth: aadlreal units swap_properties::size_units applies to (system, device, processor, memory);
  weight_units: type units (kg, g => kg * 0.001, oz => kg * 0.0283495, lb => oz * 16);
  weight: aadlreal units swap_properties::weight_units applies to (system, device, processor, memory);
  power_units: type units (W, kW => W * 1000);
  max_power: aadlreal units swap_properties::power_units applies to (system, device, processor, memory);
end swap_properties;
```

Figure 16: Case Study: AADL swap_properties.aadl

The properties are defined in the swap_properties property set are width, height, depth, weight, and max_power. Each property is assigned a domain type that constrains the properties to a domain—in our case, to the real numbers. Also, a units type is optionally assigned to each property. This constrains the units that can be used for these properties as well as conversion factors amongst other permitted units. The ability to constrain properties to particular units values as well as provide conversion factors is convenient and prevents confusion that can occur when working with unitless numbers—such as one engineer expecting units to be metric and another engineer expecting units to be imperial.

Our devices file, devices.aadl, contains processor, memory, and device SWaP properties. We only include width, height, and depth properties to the signal processing unit chassis. We also have an integration file that ties the system devices together.

```
package devices
public
    with swap_properties;

    device chassis
        properties
            swap_properties::width => 9.0cm;
            swap_properties::height => 24.0cm;
            swap_properties::depth => 19.0cm;
    end chassis;

    device main_board
        properties
            swap_properties::weight => 600.0g;
            swap_properties::max_power => 150.0W;
    end main_board;

    processor cpu
        properties
            swap_properties::weight => 150.0g;
            swap_properties::max_power => 85.0W;
    end cpu;

    memory ram
        properties
            swap_properties::weight => 30.0g;
            swap_properties::max_power => 4.0W;
    end ram;

    device network_card
        properties
            swap_properties::weight => 3.53oz;
            swap_properties::max_power => 11.0W;
    end network_card;

    device temperature_sensor
        properties
            swap_properties::weight => 0.35g;
            swap_properties::max_power => 0.5W;
    end temperature_sensor;

    device fan
        properties
            swap_properties::weight => 45.0g;
            swap_properties::max_power => 1.8W;
    end fan;

end devices;
```

Figure 17: Case Study: AADL devices.aadl with SWaP Properties

```
package integration
public
    with devices;
    system integration
    end integration;
    system implementation integration.functional
        subcomponents
            chassis: device devices::chassis;
            mb: device devices::main_board;
            cpu1: processor devices::cpu;
            ram1: memory devices::ram;
            nc: device devices::network_card;
            temp_sensor: device devices::temperature_sensor;
            fan1: device devices::fan;
            fan2: device devices::fan;
    end integration.functional;
end integration;
```

Figure 18: Case Study: AADL integration.aadl Recapitulation

We will now create an OSATE plugin to create a SWaP Analysis Report. To create the plugin, we create an Eclipse plugin project. This project contains a file named plugin.xml that sets the SWaP Analysis Report menu option as well as its command binding. We then create a class named SwapAnalysisHandler that extends the abstract class org.eclipse.core.commands.AbstractHandler. This abstract class has a method named execute that is overridden and will be invoked when the SWaP Analysis Report command is initiated by the user.

Methods and classes implemented for the SWaP Analysis Report command are as follows:

Method/Class	Description
SwapAttributes	Java Bean class that contains the constituent SWaP properties: weight, height, depth, and max power.
execute	Entry point method for the plugin.
generateCsvReport	Method to generate SWaP Analysis Report in a CSV (Comma Separated Value) format.
getSwapAttributes	Method to get the SWaP attributes for the specified property holder.
getSwapProperty	Method to get the SWaP property with the specified property name.
getSwapPropertyValue	Method to get the SWaP property value with the specified property name in the specified units.
hasSwapProperties	Method to determine if a property holder has SWaP properties.

Table 3: Case Study: AADL SWaP Analysis Report Plugin Methods/Classes

The resulting CSV loaded into Microsoft® Excel® is as follows:

	A	B	C	D	E	F
1	subcomponent	width_cm	height_cm	depth_cm	weight_kg	max_power_W
2	chassis	9	24	19	0	0
3	mb	0	0	0	0.6	150
4	cpu1	0	0	0	0.15	85
5	ram1	0	0	0	0.03	4
6	nc	0	0	0	0.1000738	11
7	temp_sensor	0	0	0	0.00035	0.5
8	fan1	0	0	0	0.045	1.8
9	fan2	0	0	0	0.045	1.8

Figure 19: Case Study: AADL SWaP Analysis Report

When gathering the SWaP properties, we were able to use an OSATE provided method PropertyUtils.getScaledNumberValue that permitted us to get property values in a specified unit irrespective of the units specified in the devices.aadl file. This allows us to compare SWaP properties more easily per our requirements specification. From the above report, we see that the dimensions of the signal processing unit chassis are 9cm x 24cm x 19cm, the weight of the signal processing unit internal components is approximately 0.97kg, and the maximum power utilized by the signal processing unit internal components is approximately 254W. These SWaP attributes are all within the allowed range per our requirements specification.

The development of the SWaP Analysis Report plugin demonstrates the power of user-defined properties in AADL and plugins in OSATE to perform analyses that are important for safety-critical systems. User-defined properties do not require changes to the core AADL language. We will now show how the core AADL can be extended with annex extensions. This will provide a means of providing more expressive syntactical structure to our model.

AADL Annex Extension For Generating Attack Trees

AADL annexes extend the core AADL language and require custom language parser implementation. Annexes are included within models using the annex keyword along with the custom annex specification language enclosed within {** and **} delimiters.

Security is of paramount importance in the analysis and design of safety-critical systems. Attack trees, introduced by Bruce Schneier, are useful conceptual diagrams showing how an asset or target might be attacked. We will create an annex specification named security_specification that will allow us to enumerate device, processor, and memory dependencies in order to create and generate an attack tree that shows the potential paths for attacking our signal processing unit.

BNF (Backus-Naur Form) notation is a formal mathematical way to describe a language and consists of a set of terminal symbols, a set of non-terminal symbols, and a set of production rules of the form (left hand side) := (right hand side). For our security_specification, we will use the following BNF:

```
LOWER_CHAR := 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
UPPER_CHAR := 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
SPECIAL_CHAR := '_'
DIGIT := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
CHAR := LOWER_CHAR | UPPER_CHAR | SPECIAL_CHAR | DIGIT
STARTING_CHAR := LOWER_CHAR | UPPER_CHAR | SPECIAL_CHAR
SUBCOMPONENT := STARTING_CHAR | SUBCOMPONENT CHAR
SUBCOMPONENT_LIST := SUBCOMPONENT | SUBCOMPONENT_LIST ',' SUBCOMPONENT
DEPENDENCY_LINE := SUBCOMPONENT ':' SUBCOMPONENT_LIST ';'
DEPENDENCIES_BLOCK := DEPENDENCY_LINE | DEPENDENCIES_BLOCK DEPENDENCY_LINE
SECURITY_SPECIFICATION := 'subcomponent_dependencies' DEPENDENCIES_BLOCK
```

Figure 20: Case Study: AADL BNF for the security_specification annex

For our case study integration file, we will add the following AADL annex security_specification:

```
annex security_specification {**
 subcomponent_dependencies
   mb : cpu1, ram1, nc, temp_sensor, fan1, fan2;
   cpu1 : mb, ram1, nc, temp_sensor, fan1, fan2;
   ram1: mb, cpu1;
   nc: mb, cpu1;
   temp_sensor: mb, cpu1;
   fan1: mb, cpu1;
   fan2: mb, cpu1;
**};
```

Figure 21: Case Study: AADL security_specification annex

We implement an Attack Tree plugin in the same way as we implemented the SWaP Analysis Report plugin in the last section. For information pertaining to OSATE plugin creation and structure, please reference the SWaP Analysis Report section preceding this section.

We parse the security_specification block, build an attack tree using a simple tree data structure, and then provide a means of generating a graphical report of the attack tree using the Java Swing graphics library.

Methods and classes implemented for the Attack Tree command are as follows:

Method/Class	Description
AttackTree	Class representing the attack tree.
execute	Entry point method for the plugin.
generateAttackTreeReport	Method to generate report containing graphical representation of the attack tree.
parseSpecification	Method to parse the security_specification per the BNF.

Table 4: Case Study: AADL Attack Tree Plugin Methods/Classes

The resulting attack tree for the security_specification for our case study results is as follows:

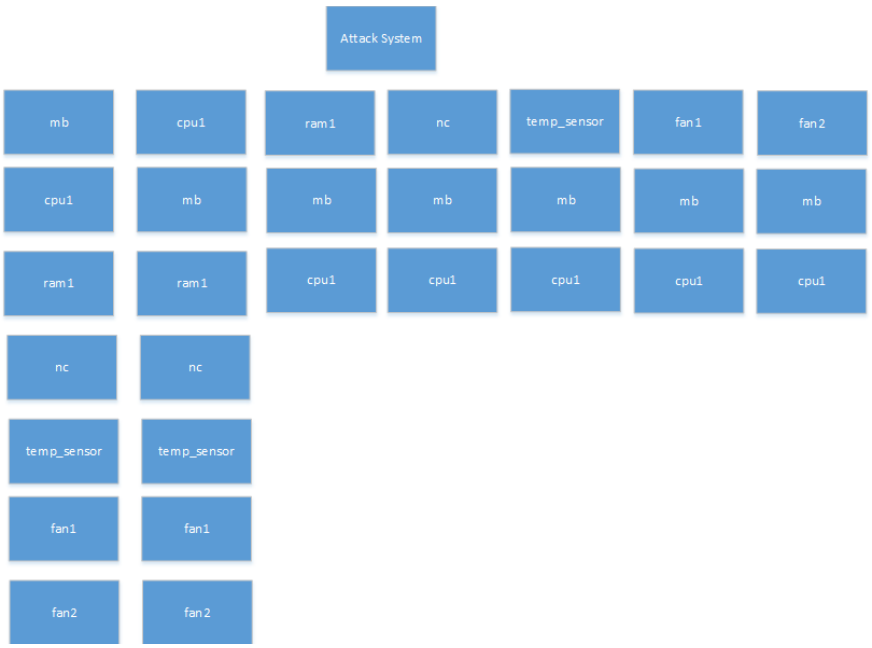

Figure 22: Case Study: AADL Attack Tree

This attack tree is simple. From any given directly exposable device, processor, or cpu within the system, the subsequent children nodes represent a chain of potential attack vectors. Please note in the generated tree that the edges have been omitted between vertices. Edges flow only vertically from the root node to the leaf node of each chain.

We have demonstrated the flexibility that AADL and OSATE provide in augmenting additional reporting and analytical capabilities to models. There are many additional applications that could be created, for instance, for DO178C certifiability or FACE (Future Airborne Capability Environment) conformance requirements that are often levied on avionics-based safety-critical systems.

Chapter 6: Related Work

OSATE is an open source platform for AADL and much work has been performed by the systems modeling community on extending the modeling language and modeling tools. We will briefly talk about projects that have been created within this space, namely: BLESS, a formal specification and verification of behaviors for embedded systems with software annex, architecture fault modeling with the AADL error-model annex, and the Cheddar plugin used for real-time scheduling analysis.

BLESS is a Behavioral Interface Specification Language (BISL) and proof environment for AADL. It is implemented as an AADL annex and introduces notations for specifying behaviors on component interfaces, defining AADL runtime aware transition systems that capture the internal behavior of AADL components—[this is similar to Java Pathfinder (JPF), a model checker for Java bytecode created by NASA that is part of the Verification and Validation course at The University of Texas at Austin], and the ability to write assertions to capture important state and event properties within the transition system notation (Larson et al., 2013).

The AADL error-model annex extends AADL to support architecture fault modeling and automated safety analysis. It provides a fault propagation ontology to support architecture fault models—focusing on fault propagation, failure behavior of individual components, and composite failure of a system in terms of its components (Delange et al., 2014).

The Cheddar plugin, started in 2002 by Frank Singhoff, is a GPL real-time scheduling tool/simulator that allows one to model software architectures of real-time systems and to check schedulability and other performance criteria—the schedulability of real-time systems can be assessed by feasibility tests or simulations (Singhoff, 2019).

Chapter 7: Conclusion

In this report, we created and modeled a simple case study and focused on extending the reporting and analysis capabilities of AADL. OSATE currently supports code generation from real-time operating systems such as VxWorks, DeOS, and POK. While the focus of AADL and OSATE are on real-time systems, this modeling language and software could also be applied to non-real-time applications that run on consumer grade operating systems such as Microsoft Windows, macOS, and GNU/Linux.

The creation of an AADL annex for the specification of unit and integration level testing would also be useful. The annex would allow specification of generated code coverage requirements (including node coverage, branch coverage, input coverage, and syntax-based coverage).

While we focused on attack trees for security analysis in AADL, a more comprehensive security suite would be beneficial. Due to the popularity of IoT (Internet of Things) and CPSs (cyberphysical systems), security is of ever increasing importance in systems analysis and software design.

FACE a standard that is becoming increasingly important to the avionics community. AADL extensions to support FACE data modeling and the auto-generation of FACE artifacts (Units of Conformance (UoC) and Units of Portability (UoP)) would be greatly beneficial to the avionics community.

Synchronization of AADL models and software running within a distributed environment would be of greatly beneficial to DevOps teams. This would allow specification of desired runtime functionality and subsequent monitoring of actual software runtime behavior to validate that systems are functioning per the specifications of the model.

References

Delange J., Feller P., "Architecture Fault Modeling with the AADL Error-Model Annex," *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, Verona, 2014, pp. 361-368, doi: 10.1109/SEAA.2014.20.

Delange, J. (2017). AADL in Practice: Become an Expert in Software Architecture Modeling and Analysis. Self-published.

Delligatti, L. (2014) SysML Distilled: A Brief Guide to the Systems Modeling Language. Addison-Wesley Professional.

INCOSE. (2007) Systems Engineering Vision 2020: INCOSE-TP-2004-004-02 2.03.

Larson B.R., Chalin P., Hatcliff J. (2013) BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software. In: Brat G., Rungta N., Venet A. (eds) NASA Formal Methods. NFM 2013. Lecture Notes in Computer Science, vol 7871. Springer, Berlin, Heidelberg.

Rumbaugh J., Jacobson I., Booch G. Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education.

Singhoff, F. (2019, June 20). Cheddar: an open-source real-time scheduling tool/simulator. Retrieved May 7, 2020, from http://beru.univ brest.fr/~singhoff/cheddar/.